ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 8, August 2016

API-Based Theoretic Metrics for Measuring the Quality of Software

R.Sowndarya, P.Subramaniam

Research Scholar, Muthayammal College of Arts & Science, Namakkal, Tamilnadu, India

Associate Professor, Dept. of Computer Science, Muthayammal College of Arts & Science, Namakkal, Tamilnadu, India

ABSTRACT: In this paper we create a set of design principles for code modularization and produce set of metrics that characterize software in relation to those principles. Some metrics are structural, architectural, and notions. The structural metrics refer to intermodal-coupling based notions. The architectural metrics refer the horizontal layering of modules in large software systems. Here we are using three types of contributions coupling, cohesion, and complexity of metrics to modularize the software. These contributions measure were primarily at the level of how the individual classes were designed from the standpoint of how many methods were packed into the classes, the depth of the inheritance tree, the inheritance fan-out, coupling between objects created by one object invoking a method on another object. Other contributions that have also used function call dependencies to characterize software modularization. Modularization algorithm is based on the combination of coupling and cohesion metrics. This is used to find modularization quality. Finally we provide two types of experiments to validate the metrics using Open-Source Software Systems. I) applied the metrics to two different versions of the same software system. II) Experimental validation consisted of randomizing a well-modularized body of software and seeing how the value of the metrics changed.

KEYWORDS: Software, Quality Measurement, algorithm;

I. INTRODUCTION

Much work has been done during the last several years on automatic approaches for code reorganization. Fundamental to any attempt at code reorganization is the division of the software into modules, publication of the API for the modules, and then requiring that the modules access each other's resources only through the published interfaces.

Our ongoing effort, from which we draw the work reported here, is focused on the case of reorganization of legacy software, consisting of millions of line of non-object-oriented code that was never modularized or poorly modularized to begin with. We can think of the problem as reorganization of millions of lines of code residing in thousands of files in hundreds of directories into modules, where each module is formed by grouping a set of entities such as files, functions, data structures and variables into a logically cohesive unit. Furthermore, each module makes itself available to the other modules through a published API. The work we report here addresses the fundamental issue of how to measure the quality of a given modularization of the software.

Note that modularization quality is not synonymous with modularization correctness. Obviously, after software has been modularized and the API of each of the modules published, the correctness can be established by checking function call dependencies at compile time and at runtime. If all intermodal function calls are routed through the published API, the modularization is correct. As a theoretical extreme, retaining all of the software in a single monolithic module is a correct modularization though it is not an acceptable solution. On the other hand, the quality of modularization has more to do with partitioning software into more maintainable modules on the basis of the cohesiveness of the service provided by each module. Ideally, while containing all of the major functions that directly contribute to a specific service vis-a-vis the other modules, each module would also contain all of the ancillary functions and the data structures if they are only needed in that module. Capturing these "cohesive services" and "ancillary support" criteria into a set of metrics is an important goal of our research. The work that we report here is a step in that direction. More specifically, we present in this work a set of metrics that measure in different ways the interactions between the different modules of a software system. It is important to realize that metrics that only analyse intermodal interactions cannot exist in isolation from other metrics that measure the quality of a given partitioning of the code. To explain this point, it is not very useful to partition a software system consisting of a couple of million lines of code into two modules, each consisting of a million lines of code, and justify the two large modules purely on the basis of function call routing through the published APIs for the two modules. Each module would still be much too large from the standpoint of code maintenance and code extension. The module interaction metrics must therefore come with a sibling set of metrics that record other desirable properties of the code.

II. PROBLEM DEFINITION

A. *Existing system*

In existing system they are used non-object oriented software system. In non-object oriented software system modularization quality is calculated only based on the modules without using classes. Theoretical validation implies conformance to a set of agreed upon principles. Code reorganization of legacy software, consisting of millions of non-object oriented code. Reorganization of millions of lines of code residing in thousands of files in hundreds of directories into modules, where each module is formed by grouping a set of entities such as file, data structures, functions and variables.

B. *Proposed system*

Proposed system is developed using Object oriented software system. Create a set design principles for code modularization and produce set of metrics. Modularization quality is calculating using metrics such as structural, architectural and notions. There are three contributions such as coupling, cohesion and complexity metrics to modularize the software. Our proposed metrics seek to characterize a body of software according to the enunciated principles. Provide two types of experiments to validate the metrics.

III. SYSTEM ANALYSIS

A. *List of Modules*

❖ **Source Code Import & Partition**

User or tester will import file/paper to our tool. The tool will partition the source code by itself.

❖ **Module Count & function Call Calculation**

In this module the tool will find the size/total number of lines in the paper. After that calculate what are the functions/methods are involved in this paper. How many methods call from other modules, how many modules call other modules and what are all the functions from other module and find how many classes and modules in a given file

❖ **Metrics Validation**

This module is heart of our paper. Here we are going to calculate the quality of software based on the modules, function and size. There are three types of metrics used to calculate the quality of software. Each metric is given various output/result. Using these outputs we can draw a graph. Finally the graph will denote the quality.

❖ **Report Generation**

In this module we are going to generate a report for our testing result. Using this report we give some suggestion to developer. There are two type reports available. One is normal report another one is graph report.

IV. Output Design

Computer output is most important and a direct source of information to the user. The output design should be in an attractive and useful form because success and acceptance of a system to some extend depends on good presentation and quality.

There are three main reasons why outputs are required. They are:

- For communicating to the persons concerned.
- For re-input to the computer for being connected with other data and further processing.
- For permanent storage
- Output devices are monitor, printer and CD Drive

A. *Database Design*

Database design is an integrated design to the file design. The general theme is to handle information as an integrated whole, with a minimum of redundancy and improved performance. Software languages are used to manipulate, describe and managed data. Regardless of the type of data structure used, the objectives of the database are:

- Accuracy and Integrity
- Successful recovery from failure
- Privacy and security of data
- Good overall performance

B. *Login Table*

FIELD NAME	DATATYPE	CONSTRAINTS
Username	Varchar(50)	Not Null
Pwd	Varchar(50)	

C. *Registration Table*

FIELD NAME	DATATYPE	CONSTRAINTS
Projcode	Int(4)	Not Null
Platform	Varchar(10)	
Bookdate	Datetime(8)	
Delivdate	Datetime(8)	
Incharge	Varchar(15)	

D. *Matrics Table*

FIELD NAME	DATATYPE	CONSTRAINTS
Paper code	Int(4)	Not Null
Module interaction	Int(4)	
Non api	Int(4)	
Api	Int(4)	
Architectural matric	Int(4)	

E. *Code Design*

The goal of coding is to translate the design of the system produced during the design phase into code in a given programming language which can be executed by a computer and that perform the computation specified by the design.Code design in this application is flexible to cope with the user and his needs. Source code clarity is enhanced by structured coding techniques, good coding style, by appropriate supporting documents, by good internal comments by the features provided in modern programming languages. The goal of the structured coding is linear size control flow through a computer program so that the execution sequence follows the sequence in which tools the code is written.

F. *Code Analyzer*

Code Analyzer is a software tool which reads any java code which uses the modularization notion. In this code analyzer tool we create four steps in menu bar as shown in figure below the steps are:-file, Find Metric Factor, Apply Formula and output the working simulation result of the following code analyzer tool is in details shown step by step.

- ***Algorithm for Metrics***
-

We are providing here the customized algorithm for the proposed metrics as per the need of the system for implementation.

Algorithm for module communication

Step 1: To find API functions of modules
Step 2: To find external calls made to the API functions
Step 3: Addition of all external calls made to the API functions
Step 4: To make total of all external calls to module
Step 5: Divide Step 3 by Step 4
Step 6: Repeat Step 1 to 5 for each module
Step 7: Add for all modules
Step 8: Divide Step 7 by total number of modules
Step 9: Output

V. IMPLEMENTATION

Implementation is the most crucial stage in achieving a successful system and giving the user's confidence that the new system is workable and effective. Implementation of a modified application to replace an existing one. This type of conversation is relatively easy to handle, provide there are no major changes in the system.

Each program is tested individually at the time of development using the data and has verified that this program linked together in the way specified in the programs specification, the computer system and its environment is tested to the satisfaction of the user. The system that has been developed is accepted and proved to be satisfactory for the user. And so the system is going to be implemented very soon. A simple operating procedure is included so that the user can understand the different functions clearly and quickly.

Initially as a first step the executable form of the application is to be created and loaded in the common server machine which is accessible to the entire user and the server is to be connected to a network. The final stage is to document the entire system which provides components and the operating procedures of the system.

VI. CONCLUSION

We have enunciated a set of design principles for code modularization and proposed a set of metrics that characterize software in relation to those principles. Although many of the principles carry intuitive plausibility, several of them are supported by the research literature published to date. Our proposed metrics seek to characterize a body of software according to the enunciated principles. The structural metrics are driven by the notion of API—a notion central to modern software development. Other metrics based on notions such as size-roundedness, size-uniformity, operational efficiency in layered architectures, and similarity of purpose play important supporting roles. These supporting metrics are essential since otherwise it would be possible to declare a malformed software system as being well-modularized. As an extreme case in point, putting all of the code in a single module would yield high values for some of the API-based metrics, since the modularization achieved would be functionally correct. We reported on two types of experiments to validate the metrics. In one type, we applied the metrics to two different versions of the same software system. Our experiments confirmed that our metrics were able to detect the improvement in modularization in keeping with the opinions expressed in the literature as to which version is considered to be better.

The other type of experimental validation consisted of randomizing a well-modularized body of software and seeing how the value of the metrics changed. This randomization very roughly simulated what sometimes can happen to a large industrial software system as new feature are added to it and as it evolves to meet the changing hardware requirements. For these experiments, we chose open-source software systems. For these systems, we took for modularization the directory structures created by the developers of the software. It was interesting to see how the changes in the values of the metrics confirmed this process of code disorganization.

REFERENCES

1. K. Sartipi.(2003) "Software Architecture Recovery Based-On Pattern Matching," PhD dissertation, School of Computer Science, Univ.Waterloo.
2. L.C. Briand, S. Morasca, and V.R. Basili.(Jan. 1996) "Property- Based Software Engineering Measurement," IEEE Trans. Software Eng., vol. 22, no. 1, pp. 68-85.
3. T.J. McCabe and A.H. Watson.(Dec. 1994) "Software Complexity," Crosstalk, J.Defense Software Eng., vol. 7, no. 12, pp. 5-9.
4. P. Oman and J. Hagemeister, "Constructing and Testing of Polynomials Predicting Software Maintainability," J. Systems and Software, vol. 24, no. 3, pp. 251-266, Mar. 1994.
5. E. Weyuker, "Evaluating Software Complexity Measures," IEEE Trans. Software Eng., vol. 14, no. 9, pp. 1357-1365, Sept. 1988.
6. S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476- 493, June 1994.